5

Derivation of an Encryption Key from a Password

In this chapter, we will learn about deriving **symmetric encryption keys** from **passwords** or **passphrases**. As we have learned in *Chapter 2*, *Symmetric Encryption and Decryption*, symmetric encryption algorithms do not encrypt with a password, but with an encryption key. Hence, in order to use a password for encryption, an encryption key must be derived from the password and then used for encryption.

This chapter features an overview of the key derivation functions supported by OpenSSL. In the practical part of the chapter, we will learn how to derive an encryption key from a password, using both the command line and C code.

We are going to cover the following topics in this chapter:

- Understanding the differences between a password and an encryption key
- What is a key derivation function?
- Overview of key derivation functions supported by OpenSSL
- Deriving a key from a password on the command line
- Deriving a key from a password programmatically

Technical requirements

This chapter will contain commands that you can run on a command line and C source code that you can build and run. For the command-line commands, you will need the `openssl` command-line tool with OpenSSL dynamic libraries. To build the C code, you will need OpenSSL dynamic or static libraries, library headers, a C compiler, and a linker.

We will implement an example program in this chapter, in order to practice what we are learning. The full source code of that program can be found here: `https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/tree/main/Chapter05`.

Understanding the differences between a password and an encryption key

What is the difference between a symmetric encryption key and a password? A symmetric encryption key is a secret array of bits directly used by an encryption algorithm. As a rule, an encryption algorithm requires an encryption key of a specific length, for example, 256 bits. Some less popular ciphers allow variable-length encryption keys, but this is an exception rather than a rule. An encryption key is not very human-friendly: it looks like random data (and often is random data), it is long to write and read, and it is impossible to memorize, unless you are superhuman.

On the contrary, a password or a passphrase is often much more human-friendly. Many passwords and passphrases are readable by a human. Especially in movies, passwords are always short, simple, and readable. No wonder people prefer 8-character passwords or 4-word passphrases over 256-bit keys, and popular encryption software provides the possibility to encrypt data with passwords instead of raw encryption keys.

While a password is more human-friendly than an encryption key, it cannot be used directly by an encryption algorithm. We have to derive a key from the password first, and then use the derived key for encryption. But how do we do it? With the help of key derivation functions!

What is a key derivation function?

A **Key Derivation Function** (**KDF**) is a function that derives a secret key of the desired bit length from some other secret material, such as a password, a passphrase, another shared secret, or a combination of asymmetric private and public keys. That other secret material is also called **Input Key Material** (**IKM**), while the secret key produced is also called **Output Key Material** (**OKM**). IKM and OKM often have different lengths. A KDF typically uses a cryptographic hash function or block cipher operations under the hood.

A **Password-Based Key Derivation Function** (**PBKDF**) is a KDF designed to produce secret keys from low-entropy IKMs, such as passwords. Those secret keys can be used as symmetric encryption keys. Another popular application of PBKDFs is password hashing. PBKDFs provide more brute-force-resistant password hashing than cryptographic hash functions alone.

Some key derivation functions are used for key exchange in secure network protocols. Those KDFs are not secure enough to derive keys from low-entropy IKMs. Hence, we are not going to review such KDFs in this chapter. We are going to focus on key derivation from passwords.

A typical KDF takes the following parameters:

- An IKM, for example, a password.
- **Salt** – some amount of randomly generated data. It is used to add more randomness to the key derivation process and hinder precomputed hash attacks, also known as rainbow table-based attacks. It is technically possible to supply empty salt to a KDF, but it is not recommended. NIST recommends salt to be at least 128 bits long. The generation of salt must not depend on IKM. Salt can be public or secret.
- **Info** – Application-specific information. It does not add security but can be useful for binding key material to its usage. Application-specific information can, for example, include the protocol version, algorithm identifier, or the user or object ID. Using different "info" values for different purposes is called **domain separation**.
- An underlying **Pseudorandom Function** (**PRF**), such as an HMAC function or a block cipher function.
- Function-specific, brute-force-resistant parameter(s). For many KDFs, it is just an iteration count, but some functions, such as *scrypt*, take several parameters influencing both *CPU* and *memory* usage.
- Desired OKM length.

Not all parameters are supported by every KDF. IKM and OKM lengths are supported by every KDF. Every good PBKDF should support salt and brute-force-resistance parameters. Some KDFs allow you to parameterize the PRF and add info.

Of all the parameters mentioned, only the IKM parameter (password) is considered secret. Other parameters are not secret and are either stored in cleartext or implied by the solution that uses the KDF, such as an authentication module or encryption software.

A secure PBKDF has the following properties:

- It is *deterministic*, meaning that given the same input parameters, the function will always produce the same secret key.
- It is *irreversible*, meaning that it is computationally infeasible to obtain the original IKM from the OKM produced. This feature is especially important for the password hashing application of KDFs because the passwords should not be easily recovered from OKMs. However, irreversibility is also important for encryption key generation, because if the same password is used to generate several encryption keys and one of those keys becomes compromised, it is not desired that the password should be recovered and used to recover other encryption keys.
- It is *brute-force* resistant, meaning that the execution of a KDF should use significant computational and memory resources.

Let's talk a bit more about brute-force resistance. It is important to understand that a typical password has much less entropy than a randomly generated secret key. In the case of a good random bit generator, a randomly generated key has as many entropy bits as its length, typically 128 or 256 bits.

Passwords are different. Currently, most websites require passwords to be at least 8 characters long. Some sites have started to require a minimum of 12 characters. Many sites demand a mixture of uppercase and lowercase Latin letters, digits, and symbols. This gives us approximately 80 possible characters. Let's assume that the probabilities of all characters to be contained in a password are uniformly distributed. Then, the entropy of a password can be calculated using the following formula:

$$\texttt{Entropy} = \log_2(\texttt{nchar}^{\texttt{plen}})$$

Where:

- `nchar` – number of possible characters, in our case 80
- `plen` – password length

According to the formula, the entropy of an 8-character password will be approximately 51 bits. If we raise the password length to 12 characters, the entropy will be approximately 76 bits.

It is important to understand that a PBKDF cannot give you more entropy than its input. If a PBKDF outputs a 256-bit key from a password containing 76 bits of entropy, and the salt is public, the output key will still have only 76 bits of entropy.

As we can see, a typical password does not have very much entropy. Therefore, PBKDFs cannot rely only on password entropy to effectively resist brute-force attacks. Password-based KDFs should also require a lot of computational resources. Many PBKDFs solve this by running a lot of iterations of the underlying PRF. At the same time, PBKDFs should not be too slow because key derivation is often used during interaction with a human, for example, during the verification of the entered password, so a PBKDF should preferably still complete in a fraction of a second.

As we can see, security level, or security strength, is not necessarily equal to the number of entropy bits. You can increase the security strength of a PBKDF by iterating the underlying PRF many times. It should be noted though that security strength does not grow linearly but logarithmically, depending on the number of iterations. For example, to increase security strength by 8 bits, you will need only 256 (2^8) iterations, but to increase security strength by 16 bits, you will need a huge 65,536 (2^{16}) iterations.

Lately, however, it has become popular to crack passwords using heavy parallelization on fast **Graphical Processing Units** (**GPUs**), **Application-Specific Integration Circuits** (**ASICs**), or **Field-Programmable Gate Arrays** (**FPGAs**), which happen to have a lot of processing power for simple computational operations. Hence, being computationally intensive is not enough for a good PBKDF. A secure PBKDF should also use a significant amount of memory, meaning that the parallelization of password cracking would be constrained by the available memory and full GPU/ASIC/FPGA processing power could not be utilized.

What PBKDFs are supported by OpenSSL? We will find out in the next section.

Overview of key derivation functions supported by OpenSSL

OpenSSL 3.0 supports several key derivation functions, but only two of them are suitable for deriving keys from passwords, namely, *scrypt* and *PBKDF2*.

PBKDF2 is a popular PBKDF, described and recommended by the PKCS #5 standard. It uses an HMAC function, such as HMAC-SHA-256, as an underlying PRF. PBKDF2 supports a tunable number of iterations and can be made computationally intensive, but not memory-intensive. In 2021, the **Open Web Application Security Project** (**OWASP**) recommended 310,000 iterations for PBKDF2 with the HMAC-SHA-256 PRF.

Scrypt is the best available choice for PBKDF in OpenSSL 3.0. Scrypt is a PBKDF that is not only computationally intensive but also memory-intensive. Scrypt uses PBKDF2 with HMAC-SHA-256 under the hood. Scrypt enables you to tune the volume of computations, memory usage, and parallelism. In 2021, OWASP recommended the following brute-force-resistant parameters for Scrypt: `N`=65,536, `r`=8, `p`=1. While Scrypt can be used to produce symmetric encryption keys, it is also one of the user password hashing methods on modern GNU/Linux systems. Another application of Scrypt is as a proof-of-work algorithm in some cryptocurrencies, such as Litecoin and Dogecoin.

Single-step KDFs and HKDF are not suitable as PBKDFs because they are not computationally intensive. They also do not take brute-force-resistant parameters.

ANSI X9.42, ANSI X9.63, and TLS1 PRFs are not computationally intensive and require TLS-specific parameters.

SSH KDF is also not computationally intensive and requires SSH-specific parameters.

Scrypt is our best choice as a PBKDF. Let's try it on the command line!

Deriving a key from a password on the command line

Producing an encryption key from a password on the command line can be done using the `openssl kdf` subcommand. This is a new subcommand that was added in OpenSSL 3.0. You can read its documentation on the `openssl-kdf` `man` page:

```
$ man openssl-kdf
```

Before deriving the key, let's generate 128-bit salt:

```
$ openssl rand -hex 16
cf0e0acf943629ecffea41c87bab94d4
```

Now we can derive a 256-bit key suitable for symmetric encryption. Let's use the *Scrypt* KDF. OWASP recommended brute-force-resistant settings and the password `SuperPa$$w0rd`:

```
$ openssl kdf \
  -keylen 32 \
  -kdfopt 'pass:SuperPa$$w0rd' \
  -kdfopt hexsalt:cf0e0acf943629ecffea41c87bab94d4 \
  -kdfopt n:65536 -kdfopt r:8 -kdfopt p:1 \
  SCRYPT
D0:3D:31:A1:A2:2A:F6:68:99:B3:02:22:60:3B:D7:21:5B:15:5B:80:2B:
85:33:36:E6:3B:AB:F9:EE:8F:FE:C7
```

Note that the command-line argument containing the password is enclosed in single quotes. This is to avoid the special interpretation of `$` characters by the command interpreter. Another method of passing special characters in a password would be to pass a hex-encoded password using the `-kdfopt hexpass:` option instead of `-kdfopt pass:`.

We have learned how to produce an encryption key from a password on the command line. Now let's learn how to do it programmatically in C code.

Deriving a key from a password programmatically

We are going to implement the `kdf` program, which will derive a key from a password.

Our key derivation program will take two command-line arguments:

1. Password
2. Hex-encoded salt

We are not going to take the `N`, `r`, and `brute-force-resistant Scrypt` parameters because because we want to simplify our example program and its usage. Instead, we are going to use OWASP-recommended settings.

OpenSSL 3.0 provides the following APIs for key derivation:

- The `PKCS5_PBKDF2_HMAC()`, `PKCS5_PBKDF2_HMAC_SHA1()`, and `EVP_PBE_scrypt()` legacy functions, specific to particular KDFs.
- The `EVP_PKEY` API. This API is intended for use with asymmetric cryptography and contains the `EVP_PKEY_derive()` function. That function is mostly intended for non-password-based key derivation during key exchange operations in secure network protocols, such as **Diffie-Hellman key exchange**, but also supports password-based key derivation using the *Scrypt* algorithm.
- The `EVP_KDF` API. This is a new API specially created for KDF, available and recommended since OpenSSL 3.0. We are going to use this API.

Documentation for the `EVP_KDF` API and its usage with *Scrypt* can be found on these man pages:

```
$ man EVP_KDF
$ man EVP_KDF-SCRYPT
```

As usual, let's make a high-level implementation plan:

1. Fetch the key derivation algorithm implementation description as an `EVP_KDF` object.
2. Create the key derivation parameters as an `OSSL_PARAM` array.
3. Create the key derivation calculation context, `EVP_KDF_CTX`.
4. Derive the key.
5. Print the produced key to `stdout`.

Now, let's implement those ideas in the code.

Implementing the kdf program

Follow these steps to implement the `kdf` program according to our plan:

1. First, we have to fetch the *Scrypt* key derivation algorithm description from OpenSSL's default algorithm provider:

   ```
   EVP_KDF* kdf = EVP_KDF_fetch(
       NULL, OSSL_KDF_NAME_SCRYPT, NULL);
   ```

2. Next, we have to create key derivation parameters:

   ```
   uint64_t scrypt_n = 65536;
   uint32_t scrypt_r = 8;
   uint32_t scrypt_p = 1;
   OSSL_PARAM params[] = {
       OSSL_PARAM_construct_octet_string(
           OSSL_KDF_PARAM_PASSWORD,
           (char*)password, strlen(password)),
       OSSL_PARAM_construct_octet_string(
           OSSL_KDF_PARAM_SALT, (char*)salt, salt_length),
       OSSL_PARAM_construct_uint64(
           OSSL_KDF_PARAM_SCRYPT_N, &scrypt_n),
       OSSL_PARAM_construct_uint32(
           OSSL_KDF_PARAM_SCRYPT_R, &scrypt_r),
   ```

```
    OSSL_PARAM_construct_uint32(
        OSSL_KDF_PARAM_SCRYPT_P, &scrypt_p),
    OSSL_PARAM_construct_end()
};
```

Unfortunately, `OSSL_PARAM_construct_*()` functions only take non-const pointers, so we have to cast `password` and `salt` pointers, and create `scrypt_n`, `scrypt_r`, and `scrypt_p` variables as non-const as well.

3. Next, we have to create the key derivation context:

```
EVP_KDF_CTX* ctx = EVP_KDF_CTX_new(kdf);
```

4. All the pieces are now in place, so it is time to derive the key:

```
EVP_KDF_derive(ctx, key, KEY_LENGTH, params);
```

5. Now that we no longer require the `EVP_KDF` and `EVP_KDF_CTX` objects, let's free them and avoid memory leaks:

```
EVP_KDF_CTX_free(ctx);
EVP_KDF_free(kdf);
```

6. The key has been produced. Let's print it in the same format as the `openssl kdf` subcommand:

```
for (size_t i = 0; i < KEY_LENGTH; ++i) {
    if (i != 0)
        printf(":");
    printf("%02X", key[i]);
}
printf("\n");
```

The complete source code for our `kdf` program can be found on GitHub as the `kdf.c` file: `https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter05/kdf.c`.

Running the kdf program

Let's run our key derivation program and see whether it works:

```
$ ./kdf 'SuperPa$$w0rd' cf0e0acf943629ecffea41c87bab94d4
D0:3D:31:A1:A2:2A:F6:68:99:B3:02:22:60:3B:D7:21:5B:15:5B:80:2B:
85:33:36:E6:3B:AB:F9:EE:8F:FE:C7
```

As we can see, the encryption key derived by our small `kdf` program is the same as the key produced by the `openssl kdf` subcommand in the previous section, suggesting that our program works correctly.

Summary

In this chapter, we have learned about the concept of deriving an encryption key from a password. Then we learned about what key derivation functions are and what the requirements are for good PBKDFs. We finished the theoretical part with a review of key derivation functions supported by OpenSSL and a recommendation on which KDF to use for password-based key derivation.

In the practical part, we learned how to derive a symmetric encryption key from a password on the command line. Then we also learned how to derive the same key programmatically in C code. We compared the resulting keys derived by both methods and, to our satisfaction, confirmed that both methods produced the same secret key.

In the next chapter, we will start learning about asymmetric cryptography.